CSCI 141
Lecture 20
Lists
Mutability
Happenings

Tech Talk: SPIE
  o Women in Software Development at SPIE
  o Wednesday, November 13th 5:30-7:00 PM in CF 115

CS Mentors Present: Debugging Workshop, Master the Art of Debugging
  o Thursday, November 14th 4:00 PM in CF 165
Viking Union - MPR
November 14
5 - 7:30 PM

Mix It Up
Science Past
The Good and The Bad

Free Food
Photobooth
Hands on Science
Story Gallery
Raffle Prizes
The Art of Salary Negotiation with Jamie Lee, Hosted by AWC
- Friday, November 15th 5:00-6:30 PM in AW 204

A study of graduating university students found that only 7% of female students attempted to negotiate an initial job offer as compared to 57% of men (Babcock & Laschever, 2003). This created a starting salary difference of 7.4%.

"...by not negotiating their job at the beginning of their career, they're leaving anywhere between $1 million and $1.5 million on the table in lost earnings over their lifetime."
Announcements
Announcements

• A4 is due tonight! Yay!
Announcements

• A4 is due tonight! Yay!

• A5 will be out this weekend, due Monday 12/2.
Announcements

• A4 is due tonight! Yay!

• A5 will be out this weekend, due Monday 12/2.
  • I'll discuss how to approach A5 in class on Monday.
Goals

• Know how to create, index, slice, and check for membership in lists.

• Understand the behavior of the +, *, in, not in, operators on lists.

• Know how to use the assignment operator on list elements and slices

• Know how to use the list methods append, and extend

• Know the definition of mutability, and which sequence types are mutable (lists) and immutable (strings, tuples)
QOTD

"To be or not to be".find("be") == 4

"Boo".replace("o", "O").lower() <= "boo"

"no" in "To be or not to be"

"stark" not in "Tony Stark"
def sub_lt(s):
    count = 0
    for i in range(len(s)):
        if s[i:] < s:
            count += 1
    return count

print(sub_lt("branStark"))
def sub_lt(s):
    count = 0
    for i in range(len(s)):
        if s[i:] < s:
            count += 1
    return count

print(sub_lt("branStark"))

<table>
<thead>
<tr>
<th>i</th>
<th>s[i:] &lt; s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>branStark &lt; branStark</td>
</tr>
<tr>
<td>1</td>
<td>ranStark &lt; branStark</td>
</tr>
<tr>
<td>2</td>
<td>anStark &lt; branStark</td>
</tr>
<tr>
<td>3</td>
<td>nStark &lt; branStark</td>
</tr>
<tr>
<td>4</td>
<td>Stark &lt; branStark</td>
</tr>
<tr>
<td>5</td>
<td>tark &lt; branStark</td>
</tr>
<tr>
<td>6</td>
<td>ark &lt; branStark</td>
</tr>
<tr>
<td>7</td>
<td>rk &lt; branStark</td>
</tr>
<tr>
<td>8</td>
<td>k &lt; branStark</td>
</tr>
</tbody>
</table>
def sub_lt(s):
    count = 0
    for i in range(len(s)):
        if s[i:] < s:
            count += 1
    return count

print(sub_lt("branStark"))
def sub_lt(s):
    count = 0
    for i in range(len(s)):
        if s[i:] < s:
            count += 1
    return count

print(sub_lt("branStark"))
def sub_lt(s):
    count = 0
    for i in range(len(s)):
        if s[i:] < s:
            count += 1
    return count

print(sub_lt("branStark"))
Lists: Yet Another Sequence Type

A list is an object that contains a sequence of values.
We've seen them before.

Values can be of any type(s)!
Lists: Yet Another Sequence Type

A list is an object that contains a sequence of values. We've seen them before.

```python
for value in [1, 16, 4]:
    print(value)
```

Values can be of any type(s)!
Lists: Yet Another Sequence Type

A list is an object that contains a sequence of values.
We've seen them before.

```python
for value in [1, 16, 4]:
    print(value)
```

Syntax:

Values can be of any type(s)!
Lists: Yet Another Sequence Type

A list is an object that contains a sequence of values. We've seen them before.

```python
for value in [1, 16, 4]:
    print(value)
```

Syntax:

```
[val0, val1, val2, val3]
```

Values can be of any type(s)!
Lists: Yet Another Sequence Type

A list is an object that contains a sequence of values.
We've seen them before.

```python
for value in [1, 16, 4]:
    print(value)
```

Syntax:

```python
[val0, val1, val2, val3]
```

comma-separated list of values

Values can be of any type(s)!
Lists: Yet Another Sequence Type

A list is an object that contains a sequence of values. We've seen them before.

```python
for value in [1, 16, 4]:
    print(value)
```

Syntax:

```
[val0, val1, val2, val3]
```

- comma-separated list of values
- surrounded by square brackets

Values can be of any type(s)!
What can we do with Lists?

A lot of this should look familiar.

These things work analogously to strings:
• Indexing
• Slicing
• The len function
• in and not in operators
• + and * operators
What can we do with Lists?

A lot of this should look familiar.

```python
a_list = ["Scott", 34, 27.7]
```

These things work analogously to strings:
- Indexing
- Slicing
- The `len` function
- `in` and `not in` operators
- `+` and `*` operators
Demo

A lot of this should look familiar.

These things work analogously to strings:
- Indexing
- Slicing
- The len function
- in and not in operators
- + and * operators
Demo

A lot of this should look familiar.

```
a_list = [ "Scott", 34, 27.7 ]
```

These things work analogously to strings:

- Indexing
- Slicing
- The len function
- in and not in operators
- + and * operators
Demo

A lot of this should look familiar.

make 'em

index 'em

index 'em

slice 'em
Demo

A lot of this should look familiar.

```python
a_list = ["Scott", 34, 27.7]  # make 'em
a_list[0]  # index 'em
a_list[-1]  # index 'em
a_list[1:]  # slice 'em
```
Demo
Demo

```python
a_list = ["Scott", 34, 27.7]
len(a_list)
len(['abc'])
len([])
34 in a_list
"34" not in a_list
a_list + ['Wehrwein', 'WWU']
['na'] * 16 + ['Batman']
a_list[0:2]
a_list[0]  # this is an element of the list
a_list[0:1]  # this is a length-1 list!
# slices always give you back a list.
```
What can go in lists?

• Like tuples, *any* value can go in a list.
  
  • tuples, lists, Turtles, ... *anything*
Demo

Lists can contain any type: lists, tuples, turtles, ...
Demo

Lists can contain any type: lists, tuples, turtles, ...

```python
a_list = ["Scott", [34, 27.7, (39, 70)]]
a_list[0]
a_list[1]
a_list[1][2]
a_list[1][2][0]
```
Lists: Lightning Round!

True or False?

starks = ["Ned", "Arya", "Bran", "Sansa"]
Lists: Lightning Round!

True or False?

```
starks = [ "Ned", "Arya", "Bran", "Sansa"]

"Ned" in starks
```
Lists: Lightning Round!

True or False?

starks = ["Ned", "Arya", "Bran", "Sansa"]

✓ "Ned" in starks
Lists: Lightning Round!

True or False?

starks = [ "Ned", "Arya", "Bran", "Sansa"]

✓ "Ned" in starks

"Sansa" in starks[1:3]
Lists: Lightning Round!

True or False?

starks = ['Ned', 'Arya', 'Bran', 'Sansa']

✓ "Ned" in starks

✗ "Sansa" in starks[1:3]
Lists: Lightning Round!

True or False?

starks = ["Ned", "Arya", "Bran", "Sansa"]

✓ "Ned" in starks

✗ "Sansa" in starks[1:3]

\[\text{len(starks[1:4]) == 3}\]
Lists: Lightning Round!

True or False?

starks = ["Ned", "Arya", "Bran", "Sansa"]

✓ "Ned" in starks

✗ "Sansa" in starks[1:3]

✓ len(starks[1:4]) == 3
Lists: Lightning Round!

True or False?

starks = ["Ned", "Arya", "Bran", "Sansa"]

✓ "Ned" in starks

✗ "Sansa" in starks[1:3]

✓ len(starks[1:4]) == 3

✓ "Arya" in (starks + ["Jon"]) [2:]
Lists: Lightning Round!

True or False?

starks = ["Ned", "Arya", "Bran", "Sansa"]

✓ "Ned" in starks

✗ "Sansa" in starks[1:3]

✓ len(starks[1:4]) == 3

✗ "Arya" in (starks + ["Jon"]) [2:]
Lists: Lightning Round!

True or False?

```
starks = ["Ned", "Arya", "Bran", "Sansa"]

✓ "Ned" in starks

✗ "Sansa" in starks[1:3]

✓ len(starks[1:4]) == 3

✗ "Arya" in (starks + ["Jon"]) [2:]

len(starks[1:2] * 4) == 8
```
Lists: Lightning Round!

True or False?

starks = ["Ned", "Arya", "Bran", "Sansa"]

✓ "Ned" in starks

✗ "Sansa" in starks[1:3]

✓ len(starks[1:4]) == 3

✗ "Arya" in (starks + ["Jon"]) [2:]

✗ len(starks[1:2] * 4) == 8
Lists vs Strings: What's the difference?

1. Strings hold only characters, while lists can hold values of any type(s).
Lists vs Strings: What's the difference?

1. Strings hold only characters, while lists can hold values of any type(s).

...haven't we seen this before?
Lists vs Strings: What's the difference?

1. Strings hold only characters, while lists can hold values of any type(s).

   ...haven't we seen this before?

**Tuples** are also objects that hold a sequence of values of any type(s).
Lists vs Strings: What's the difference?

1. Strings hold only characters, while lists can hold values of any type(s).

   ...haven't we seen this before?

**Tuples** are also objects that hold a sequence of values of any type(s).

   ("alpaca", 14, 27.6)
Lists vs Tuples: What's the difference?

**Tuples** are *also* objects that hold a sequence of values of any type(s).
Lists vs Tuples: What's the difference?

**Tuples** are *also* objects that hold a sequence of values of any type(s).

**Tuples** are **immutable**: their contents **cannot** be changed.
Lists vs Tuples: What's the difference?

**Tuples** are *also* objects that hold a sequence of values of any type(s).

**Tuples** are immutable: their contents **cannot** be changed.

**Lists** are mutable: their contents **can** be changed.
Lists vs Tuples: What's the difference?

**Tuples** are *also* objects that hold a sequence of values of any type(s).

**Tuples** are immutable: their contents **cannot** be changed.

**Lists** are mutable: their contents **can** be changed.

```python
a_tuple = ("a", 14, 27.6)
```
Lists vs Tuples: What's the difference?

**Tuples** are also objects that hold a sequence of values of any type(s).

**Tuples** are **immutable**: their contents **cannot** be changed.

**Lists** are **mutable**: their contents **can** be changed.

```python
a_tuple = ("a", 14, 27.6)
a_list = ["a", 14, 27.6]
```
Lists vs Tuples: What's the difference?

**Tuples** are *also* objects that hold a sequence of values of any type(s).

**Tuples** are *immutable*: their contents *cannot* be changed.

**Lists** are *mutable*: their contents *can* be changed.

```python
a_tuple = ("a", 14, 27.6)
a_list = ["a", 14, 27.6]

a_tuple[1]  # => 14
```
Lists vs Tuples: What's the difference?

**Tuples** are *also* objects that hold a sequence of values of any type(s).

**Tuples** are **immutable**: their contents **cannot** be changed.

**Lists** are **mutable**: their contents **can** be changed.

```python
a_tuple = ("a", 14, 27.6)
a_list = ["a", 14, 27.6]

a_tuple[1] # => 14
a_list[1] # => 14
```
Lists vs Tuples: What's the difference?

**Tuples** are also objects that hold a sequence of values of any type(s).

**Tuples** are immutable: their contents **cannot** be changed.

**Lists** are mutable: their contents **can** be changed.

```python
a_tuple = ("a", 14, 27.6)
a_list = ["a", 14, 27.6]

a_tuple[1]  # => 14
a_list[1]   # => 14

a_tuple[1] = 0  # causes an error
```
Lists vs Tuples: What's the difference?

**Tuples** are also objects that hold a sequence of values of any type(s).

**Tuples** are immutable: their contents **cannot** be changed.

**Lists** are mutable: their contents **can** be changed.

```python
da_tuple = ("a", 14, 27.6)
da_list = ["a", 14, 27.6]

da_tuple[1]  # => 14
a_list[1]   # => 14

da_tuple[1] = 0  # causes an error
a_list[1] = 0  # a_list is now ["a", 0, 27.6]
```
Lists vs Tuples: What's the difference?

**Tuples** are also objects that hold a sequence of values of any type(s).

**Tuples** are **immutable**: their contents **cannot** be changed.

**Lists** are **mutable**: their contents **can** be changed.

```python
a_tuple = ("a", 14, 27.6)
a_list = ["a", 14, 27.6]

a_tuple[1]  # => 14
a_list[1]   # => 14

a_tuple[1] = 0  # causes an error
a_list[1] = 0   # a_list is now ["a", 0, 27.6]
```
Lists vs Tuples: What's the difference?

**Tuples** are also objects that hold a sequence of values of any type(s).

**Tuples** are **immutable**: their contents **cannot** be changed.

**Lists** are **mutable**: their contents **can** be changed.

```python
a_tuple = ("a", 14, 27.6)
a_list = ["a", 14, 27.6]

a_tuple[1]  # => 14
a_list[1]   # => 14

a_tuple[1] = 0  # causes an error
a_list[1] = 0   # a_list is now ["a", 0, 27.6]
```
Lists are mutable

```python
a_list = ['a', 14, 27.6]
```

```markdown
a_list ← ['a', 14, 27.6]
```
Lists are mutable

```python
a_list = ["a", 14, 27.6]
a_list[0] = "b"
```

```python
a_list # ["b", 14, 27.6]
```
Lists are mutable

```python
a_list = ["a", 14, 27.6]
a_list[0] = "b"
a_list.append(19)
```

```
["b", 14, 27.6, 19]
```
Lists are mutable

```python
a_list = ["a", 14, 27.6]
a_list[0] = "b"
a_list.append(19)
a_list.append(["12", 2])
```

```
[ "b", 14, 27.6, 19, [ "12", 2 ] ]
```
Lists are mutable

a_list = ["a", 14, 27.6]

a_list[0] = "b"

a_list.append(19)

a_list.append(["12", 2])

a_list.extend([22, 33])

a_list

["b", 14, 27.6, 19, ["12", 2], 22, 23]
Lists are mutable

Notice the difference between string methods and list methods:

```python
new_string = a_string.lower()
a_list.append(19)
```

```
["b"]
```

```
"JON"
a_string
```
Lists are mutable

Notice the difference between string methods and list methods:

```python
a_list.append(19)  # modifies the list in-place
# a_list = ["b", 19]
```

```python
new_string = a_string.lower()  # has no return value
# a_string = "JON"
```
Lists are mutable

Notice the difference between string methods and list methods:

```python
a_list.append(19)  # modifies the list in-place
# has no return value

new_string = a_string.lower()  # does not modify a_string
# returns a lower-case copy
```

- `a_list.append(19)` modifies the list in-place.
- Has no return value.

- `new_string = a_string.lower()` does not modify `a_string`.
- Returns a lower-case copy.
List Mutability and Methods

```python
a = ["Abe", "Ike"]
a.append("JFK")
a.extend(["FDR", "Geo"])
a[0] = a[:2]
print(a)
```

A. ["Abe", "Ike", "JFK", ["FDR", "Geo"]]
B. ["Abe", "Ike", "JFK", "FDR", "Geo"]
C. [["Abe", "Ike"], "Ike", "JFK", "FDR", "Geo"]
D. ["Abe", "Ike", "Ike", "JFK", "FDR", "Geo"]
List assignment + slicing
List assignment + slicing

We can assign to indices:
List assignment + slicing

We can assign to indices:

\[ a = [5, 6, 7, 8] \]
\[ a[0] = 10 \]
List assignment + slicing

We can **assign** to indices:

\[
a = [5, 6, 7, 8]
\]

\[
a[0] = 10
\]

We can **slice** out sublists:
List assignment + slicing

We can **assign** to indices:

\[
a = [5, 6, 7, 8]
a[0] = 10
\]

We can **slice** out sublists:

\[
a[0:3] \ # \Rightarrow [5, 6, 7]
\]
List assignment + slicing

We can **assign** to indices:

```python
a = [5, 6, 7, 8]
a[0] = 10
```

We can **slice** out sublists:

```python
a[0:3] # => [5, 6, 7]
```

Can we **assign** to **slices**?
List assignment + slicing

We can **assign** to indices:

\[
a = [5, 6, 7, 8]
\]
\[
a[0] = 10
\]

We can **slice** out sublists:

\[
a[0:3] \# \Rightarrow [5, 6, 7]
\]

Can we **assign** to slices?

**You betcha!** (demo)
List assignment + slicing: Demo

a = [5, 6, 7, 8]
a[::2] = [3, 4]

a = [5, 6, 7, 8]
a[::3] = a[1::]

a = [5, 6, 7, 8]
a[::2] = a[1::]
Demo: What are lists good for?

• Generate a list of the fibonacci sequence
  • fib_list.py

• Make a deck of cards and deal a blackjack hand
  • blackjack.py

• Make a *bale* of turtles do some crazy stuff.
  • bale.py
Demo: a *bale* of turtles

- bale.py